

Faster Exact Search using Document Clustering

Jonathan Dimond and Peter Sanders
 Karlsruhe Institute of Technology, Karlsruhe, Germany
 sanders@kit.edu, mail@dimond.de

Thursday 6th November, 2014

Abstract

We show how full-text search based on inverted indices can be accelerated by clustering the documents without losing results (SeCluD – **S**earch with **C**lustered **D**ocuments). We develop a fast multilevel clustering algorithm that explicitly uses query cost for conjunctive queries as an objective function. Depending on the inputs we get up to four times faster than non-clustered search. The resulting clusters are also useful for data compression and for distributing the work over many machines.

1 Introduction

Full-text search is one of the enabling techniques of the information society since it is needed for all kinds of search engines. The approach most used in practice uses *inverted indices*: Consider a set D of n documents. Each document contains a set of *terms* from a dictionary T . The index stores for each term $t \in T$ a *posting list* of document IDs where it occurs. A typical query asks for documents containing a set of two or more terms. This query can then be answered by intersecting the corresponding posting lists. Unfortunately, for large inputs, the lists become huge incurring substantial energy consumption for full-text search. For example, for web search, thousands of machines can be involved in answering a single query. What helps is that two sorted sequences of document IDs can be intersected in time linear in the size of the *smaller* list [14].

The starting point for this paper was the observation that we can boost the impact of such advanced set intersection algorithms by distributing the documents to clusters where terms are distributed as nonuniformly as possible. Lets consider a simple example to illustrate this effect. Suppose we want to know all documents that contain both term a and term b . Assume we have four clusters with the following number of occurrences of a and b .

Cluster	# a	# b	min
1	2 000	10 000	2 000
2	10 000	1 000	1 000
3	40 000	1 000	1 000
4	1 000	25 000	1 000
Σ	53 000	37 000	5 000

Counting $\min(x, y)$ steps of the algorithm from [14] for intersecting lists of size x and y respectively, we get a total of 5 000 steps for performing the intersection in all clusters whereas we get 37 000

steps for intersecting the posting lists in an unclustered scenario – more than a factor five difference. More generally, for conjunctive queries where the number of term occurrences in the clusters is not too highly correlated, we expect improved performance from clustering. The main purpose of this paper is to get a first idea how much this approach could help in practice. After introducing basic concepts in Section 2, we formalize this idea in Section 3 and develop an efficient clustering algorithm for the resulting objective function. This algorithm is based on the well known K-Means principle but uses a fast multilevel scheme for initialization that may be of independent interest. In Section 5 we evaluate our approach using large real world instances. Section 6 discusses the results and possible future work.

More Related Work

This paper is based on the diploma thesis of Jonathan Dimond [6]. Clustering has been previously proposed for accelerating full-text search, e.g., [10, Section 7.1.6] following the *clustering hypothesis* [16] – documents that are relevant to a query tend to be more similar to each other than documents that are not relevant. Documents similar to each other are clustered together. Queries are then executed via *collection selection*. Document clusters considered irrelevant are disregarded and only clusters relevant to the query are used for searching. Although this yields good scores in standard information retrieval benchmarks, such an approach may lead to unexpected results in practice since unsupervised learning techniques such as document clustering are notoriously unreliable.

In many commercial applications one even wants – at least as a first step – a complete and well defined set of results. For example, in SAP HANA [8, 15] the default mode of full-text search to is to find *all* documents matching a query. One reason is that the full text query is often only one of several filtering criteria in a complex SQL query. Further speedup techniques not based on document clustering have also been considered including geographical tiering [3], static index pruning [5] and dynamic index pruning [13]. However, all these techniques improve efficiency by disregarding parts of the index.

2 Preliminaries

Let $D = \{d_1, \dots, d_n\}$ denote the set of documents and $N = \sum_{d \in D} |d|$ the total size of the corpus. Furthermore, let $T = \{t_1, \dots, t_m\}$ denote the set of terms occurring in D . Given a desired number of clusters k , we want to partition the documents into a set of cluster $C = \{c_1, \dots, c_k\}$ such that the expected query time is small.

3 Clustering Based Search

3.1 Developing an Objective Function

Of course, query costs depend on the query algorithm and the distribution of queries. In order to come to an easy to handle objective function, we make some assumptions and simplifications here that arguably lead to little loss in precision.

First of all, we focus on exact conjunctive queries involving exactly two terms. Exact conjunctive queries with more than two terms are covered insofar as it is usually a good idea to first intersect the lists for the two most rare terms and often this takes most of the query time. We ignore ranking

in this paper for simplicity and because some applications require computing the full set of answers, for example in a relational database context where further filtering may throw away most of the results later.

In our implementation, we assume a two-level query algorithm that first inspects an inverted index where each cluster is viewed as one document and then inspects each cluster containing any document with the search terms. See [6] for details. To simplify the exposition, in this section, we assume that the query cost is the sum of the per cluster query costs. Let $\Phi(x, y)$ denote the cost for intersection two lists of length x and y respectively. For the Lookup algorithm from [14] we can approximate $\Phi(x, y) = \min(x, y)$. For a particular query (t, u) we then get cost

$$\Psi_C(t, u) := \sum_{i=1}^k \Phi(n_i(t), n_i(u))$$

where $n_i(t)$ is the number of documents in cluster i of clustering C containing term t . When the clustering C is clear from the context, we omit the subscript ‘ C ’.

If we know the distribution of queries, we can now compute the expected cost of a query:

$$\psi := \mathbf{E}[\Psi] := \sum_{\{t, u\} \in \binom{T}{2}} \mathbf{P}[(t, u)] \Psi(t, u) \quad (1)$$

where $\mathbf{P}[(t, u)]$ denotes the probability of observing query (t, u) . Unfortunately, it is unrealistic to assume that we know the probability of all queries so that we need an approximation. We thus only assume that we know the probability of each query term. These probabilities can be estimated from a query log or (less accurately) using statistics on the frequency of the terms in the document collection itself. If we now also assume that terms in a query are chosen independently, we get

$$\psi = \sum_{\{t, u\} \in \binom{T}{2}} \mathbf{P}[t] \mathbf{P}[u] \Psi(t, u) \quad (2)$$

3.2 The Clustering Algorithm

Our starting point is an adaptation of the well known K-means algorithm to our problem: In each iteration of the algorithm, each document d is added to the cluster where it “fits” best. In order to decide what the best fit is for objective function ψ from Equation (2) we only have to decide how ψ changes when d is added. Hence, let $\delta_j^+(d)$ denote the change in ψ when document d is added to cluster j . Similarly, let $\delta_j^+(t)$ denote the change in ψ when a document is added to cluster j that contains only the single term t . We have $\delta_j^+(d) = \sum_{t \in d} \delta_j^+(t)$. Adding a single term t to cluster j only affects a summand $\mathbf{P}[t] \mathbf{P}[u] \min(n_j(t), n_j(u))$ of ψ if $n_j(t) < n_j(u)$. In that case, $\min(n_j(t), n_j(u))$ increases by one. Hence,

$$\delta_j^+(t) = \mathbf{P}[t] \sum_{u \neq t, n_j(t) < n_j(u)} \mathbf{P}[u] \quad .$$

$\delta_j^+(t)$ can be computed in constant time using a lookup table that has to be recomputed after each iteration. Hence, $\delta_j^+(d) = \sum_{t \in d} \delta_j^+(t)$ can be computed in time linear in the document size. Computing this lookup table itself can be done by sorting the terms occurring in cluster j by $n_j(t)$ and then computing a prefix sum over that array. Assuming that the term frequencies are polynomial in the number of terms occurring in a cluster, sorting can be done in linear time. Overall, one iteration of the K-means algorithm then takes time (and space) $\mathcal{O}(kN)$.

Refinements

The basic clustering algorithm defined above is already quite fast. However a number of additional improvements are critical to scale to really large inputs.

Multilevel Initialization. K-Means algorithms converge much faster if the initial solution is already of high quality. We use a multilevel initialization that may be of independent interest also in other applications. For a scaling factor $\epsilon < 1$, we take a sample of size $\max(k, \epsilon|D|)$ of the documents, cluster it recursively into k clusters (which is trivial for the base case $|D| = k$) and then run the K-means algorithm. The K-means algorithm can lead to oscillations in the clustering that are particularly pronounced when the clusters are small. When D becomes small, we therefore switch to an algorithm that updates the objective function after every assignment of a document.

TopDown Clustering. Since the running time of the K-Means algorithm grows at least linearly with the number of clusters k , we use a hierarchical clustering algorithms that recursively splits the documents: Subproblems with $s > |D|/k$ documents are split into $\min(\chi, sk/|D|)$ pieces where the *splitting factor* χ is a tuning parameter. This way we obtain between k and $2k$ clusters. An important side effect is that this approach balances cluster sizes.

Ignoring Infrequent Terms. Most search time is spent on queries involving long posting lists, i.e., frequent terms. Hence, the rare terms hardly contribute to the overall cost of queries. Therefore, we can ignore rare terms while evaluating the objective function without significantly affecting overall performance. This greatly accelerate the clustering algorithm and simplifies its parallelization.

Parallelization. Our implementation uses shared memory parallelization. A massively parallel distributed memory implementation is also easy as long as we can afford to store word frequency statistics for each cluster and each frequent term on each node of the system – simply assign a subset of the documents to each node. In connection with the above optimizations on TopDown clustering and ignoring infrequent terms, this seems quite realistic even for huge document collections.

3.3 Query Algorithms

As already explained, we focus on conjunctive queries involving two terms. The most direct way to use the clustering is to run the query on each cluster. When the number of clusters k is large, a possible improvement is to build a *cluster index* listing for each term which clusters contain documents with this term. A query (t, u) will then first intersect the list corresponding to t and u in the cluster index. The query is then only forwarded to the clusters containing both t and u . Note that a cluster index can be viewed as an inverted index for a corpus with k documents where each documents is the concatenation of all the documents in a cluster.

We can also use the clustering to reorder the documents: The j -th document in cluster i gets document id $j + \sum_{\ell < i} |c_\ell|$. Beyond this reordering, the clustering is ignored – we use the single-cluster Lookup algorithm. The motivation for this is the observation in [14] that nonuniform distribution of terms in the documents actually accelerates the set intersection. Renumbering makes the distribution less uniform and thus may accelerate the search.

4 Implementation Details

We have made a prototypical implementation using about 4 000 lines of Haskell and 2 000 lines of C where all the time critical parts – clustering and query – are implemented in C. We use OpenMP for parallelization¹. We switch to document grained updates of the objective function once $|D| < 100k$. The default shrink factor for the multilevel initialization is $\epsilon = 0.1$. The K-means algorithm repeats as long as the objective function value improves by at least 1 %. The splitting factor for hierarchical clustering is set to $\chi = 8$. Only the $TC = 100\,000$ most frequent terms are used for clustering. The Lookup algorithm [14] uses bucket size 16 (8 for the cluster index). This seems a good tradeoff between space and speed requirements.

5 Experiments

	GOV2	GOV2s	Wikipedia	pagenstecher.de
Documents	25 205 179	631 975 969	6 096 279	786 474
Terms	38 562 580	25 221 691	12 295 297	573 725
Terms / Document	652.22	18.19	230.54	35.70
Input size (raw)	396.74 GB	396.74 GB	12.37 GB	175.14 MB
Inverted Index size	16.25 GB	32.83 GB	3.09 GB	89.8 MB

Table 1: Dataset Statistics

	AOL	Wikipedia	pagenstecher.de
Queries	29 077 553	11 000 000	13 230
Distinct Terms	1 501 946	1 067 091	981

Table 2: Query Log Statistics

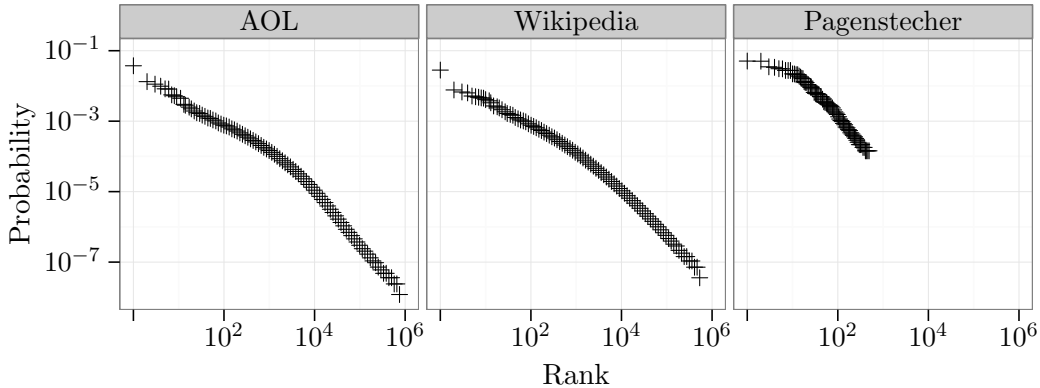


Figure 1: Probability of a term appearing in a query as a function of its rank on a log-log scale. A sample of 100 terms with exponentially growing ranks is plotted.

¹Source are here: <https://www.github.com/jdimond/diplomarbeit>

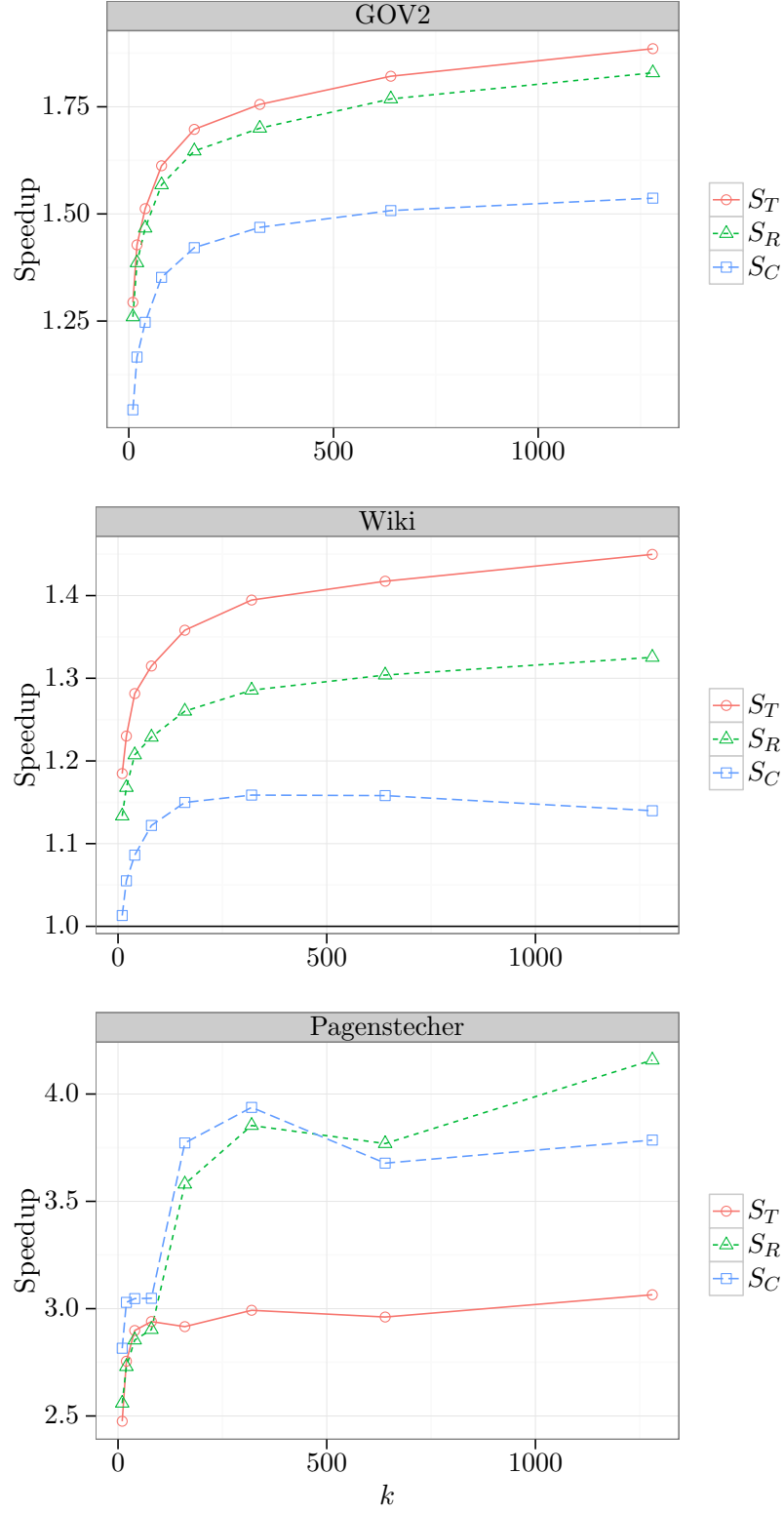


Figure 2: Speedups for different number of clusters (flat clustering).

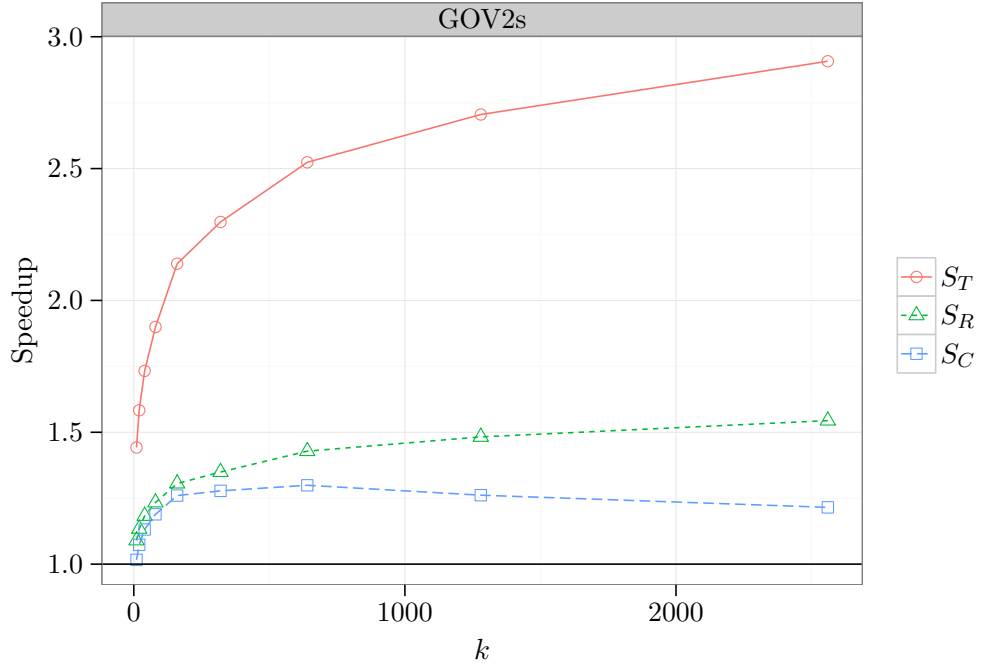


Figure 3: Speedups for different number of clusters (TopDown clustering).

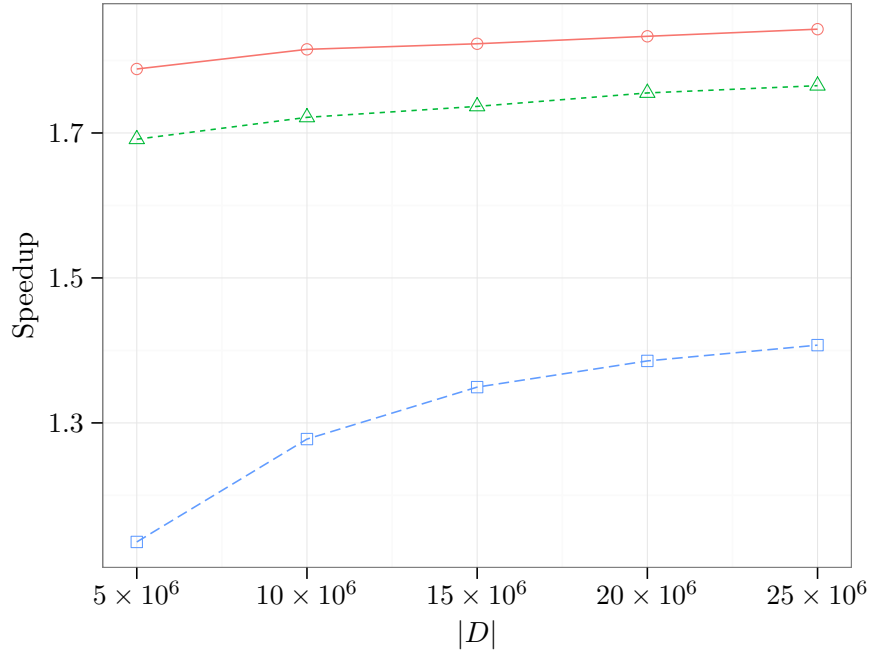


Figure 4: Speedups for GOV2 with varying number of documents $|D|$ for $k = 2500$ clusters.

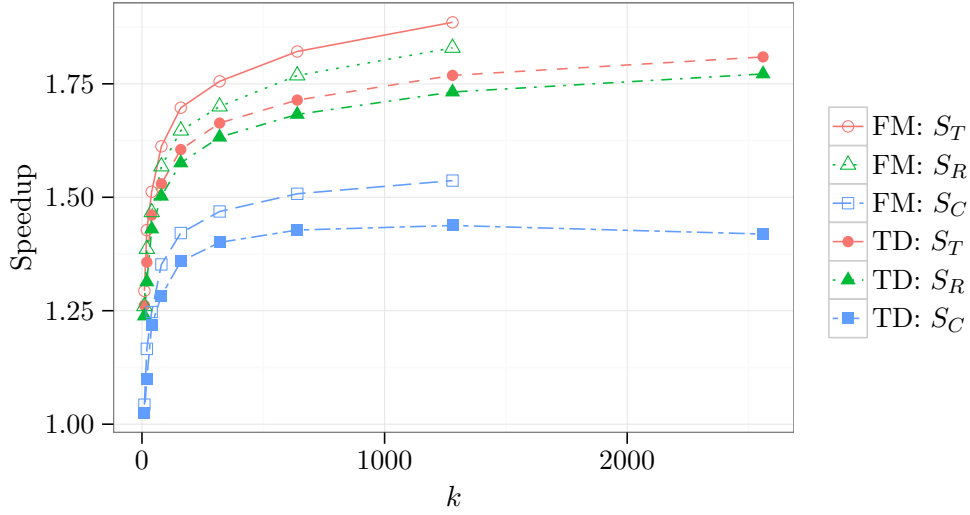


Figure 5: Speedups comparison for flat (FM) and TopDown (TD) clustering (GOV2).

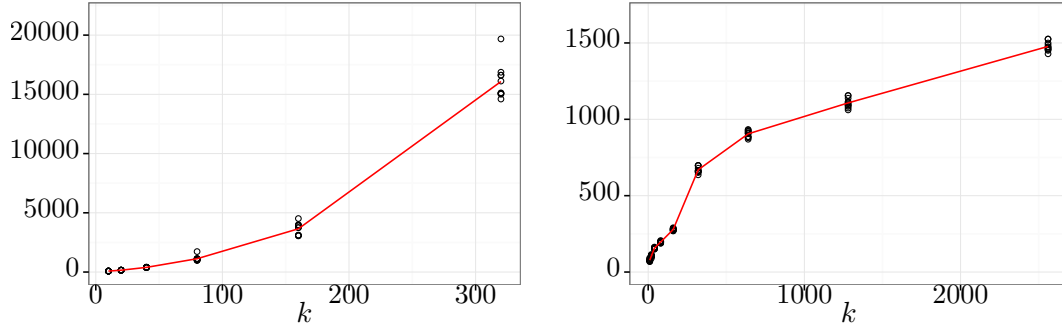


Figure 6: Clustering times [s] for flat (left) and TopDown (right) clustering for 10 independent trials.

All experiments were done on a machine with two octa-core Intel Sandy Bridge Xeon E5-2670 processors with 2.6GHz and 64 GB RAM (i.e., 16 cores and 32 hardware threads). The operating system was SuSE Linux Enterprise Server 11 (kernel version 3.0.42). The compilers used were GHC 7.6.2 and GCC 4.7.2 with optimization level `-O3`. Clustering is run in parallel. Queries are run sequentially.

Table 1 gives the text corpora used for our benchmarks. GOV2 [4] is one of the standard benchmarks used in the literature. GOV2s is the same corpus but each *sentence* is used as one document. The sentences were extracted using the Stanford NLP library [11]. This emulates a corpus with many very small documents as you may find it in corporate databases and a situation where you are looking for terms occurring together in the same sentence. Wikipedia is the plain text contained in the articles in the English Wikipedia in May 3, 2013. Pagenstecher.de contains user posts from a German online community for car tuning. This corpus was selected because we have an authentic query log for it.

Further query logs used are shown in Table 2. The AOL log contains the two term queries from

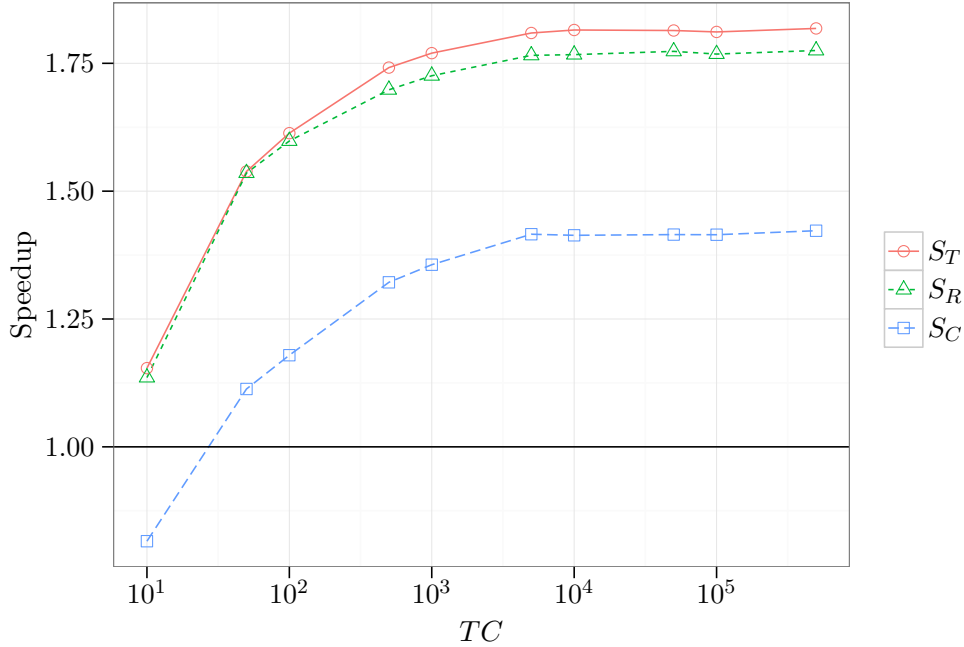


Figure 7: Speedups for GOV2 with varying TC for $K = 2500$.

[12]. We use this log as semi-realistic input for our GOV2 and GOV2s test collections. For the Wikipedia corpus, we generate a synthetic log: for each article reference, we add all pairs of terms in the title of the referenced article to the log. Figure 1 shows the distribution of term frequencies in these logs. We can see that all of them, including the synthetic Wikipedia log, show a Zipf-like distribution of term frequencies.

Our main concern are speedups over the single cluster case. We distinguish the “theoretical” speedup S_T predicted by the function ψ from Equation (2), the speedup S_C obtained using the cluster index from Section 3.3, and the speedup S_R using the reordering algorithm from Section 3.3. Figures 2 and Figures 3 compare these values for varying number k of clusters. Most of the time, S_T overestimates the speedup but it seems strongly correlated with the actual behavior – which is all we need for making it a useful objective function for the clustering algorithm. Generally, increasing the number of clusters helps to increase speedup. However, for the Wikipedia instance using the cluster index, the speedup decreases for $k > 500$ – it is clear that eventually, overheads for the additional indirection start to show. A surprise is that the reordering algorithm achieves much better performance than the cluster index. Overall, we achieve speedups between 1.3 and 4 which is not overwhelming but certainly significant and possibly useful. The Pagenstecher instance seems very different from the others – it achieves much higher speedups and it is the only one where the theoretical speedup is smaller than the practical speedups. This may simply be due to its relatively small size or specialized topic but it is also the only one with truly realistic query logs. If it turns out that this is the reason for the performance difference, we might hope for larger speedups also for large instances. Figure 4 gives some reason for optimism here since it indicates that speedups may actually *increase* with growing number of documents.

Figure 5 indicates that the flat clustering algorithm gives slightly better speedups than the TopDown algorithm. However, for the values of k that give good speedup, we do not really have a choice – Figure 6 shows that the running time of the flat clustering algorithms actually grows *superlinearly* with the number of clusters k – apparently, the algorithm also converges more slowly for large k . The TopDown algorithm is orders of magnitude faster than the flat algorithm. Indeed, our clustering algorithm needs much less time than the time for parsing and indexing the documents. Hence, the preprocessing overhead is not a big issue.

Usually, preprocessing techniques also come with a penalty for storing the preprocessed information. However, in our case the contrary is the case here – clustering allows better compression of the posting lists, see Appendix A for details.

It might be argued that it is risky that our objective function is tied to a particular intersection algorithm. To assess this risk we have evaluated the theoretical speedup for a different cost function that assumes an asymptotically optimal comparison based intersection algorithm with running time $\Phi(x, y) = x \log \frac{y}{x}$ for $x > y$ (e.g., [1]). Appendix B indicates that this gives very similar results – indeed, the theoretical speedups are even higher than for the lookup algorithm even though $\min(x, y)$ was used as the objective function for clustering.

In Figure 7 we investigate how the number of terms (TC) we use for clustering influences speedup. All three speedup measure show that even the 10000 most frequent terms would be enough for the GOV2 input. This is good news since this allows faster clustering algorithm. In particular, a massively parallel clustering algorithm can probably afford to replicate all lookup tables over all processors.

6 Conclusions

We have demonstrated that document clustering can significantly accelerate conjunctive queries while still giving exact results. Using a multilevel hierarchical clustering algorithm we were able to do high quality clustering even faster than the needed for parsing and indexing the documents. Our approach of tailoring the objective function for clustering to the actual performance the query algorithm might also be useful in other situations like clustering for inexact search or in order to compress data.

Several ideas suggest themselves for further improving the results. Since the approach is most useful for big inputs, scaling to even bigger corpora may be the main concern. This seems feasible since a massively parallel implementation is relatively easy. We have not done so yet since it naturally requires a lot of resources.

Equally interesting are efforts to increase the quality of the clustering. Currently, we only look at the impact of adding documents to a cluster. However, for small clusters (e.g., during initialization) it also matters how removing a document from a cluster affects query performance. We believe that our lookup-table approach can take this into account without big performance penalties. Also, compromises between the pure round based K-means algorithm and the variant with document-wise updates seem possible which might improve convergence speed and overall quality. Also further tuning of the TopDown algorithm seems promising. Probably one wants to use larger splitting factors at least at the top of the recursion tree in order to improve quality.

Since using the clustering for reordering posting lists was very successful, this should also be considered more closely. In particular, we would like to have a more symmetric version of the lookup algorithm for list intersection. Currently, the algorithm traverses the shorter list while

making lookups in the longer list. Actually, we would like to have a more adaptive algorithm that scans the list which is more dense at the current position. For example, when a lookup finds an empty bucket, we might switch to the other list. Also, it is quite clear that for clustering for reordering should (recursively) go all the way down to clusters consisting only of a few documents. To make this efficient, we need to dynamize our cutoffs – we only want to consider terms frequent in the remaining documents.

Last but not least, we have to investigate what happens for other types of queries. Perhaps most interesting are top-K queries where we are only interested in the most relevant results using some scoring function. Once more, we want to exploit the information inherent in the clustering to gain performance but the clustering must not influence the result. The hope might be that a cluster c which contains only few documents with a certain query term then c might contain even less *relevant* documents with that term so that only a fraction of its posting list actually needs to be considered.

References

- [1] R. Baeza-Yates. A fast set intersection algorithm for sorted sequences. In *15th Symposium on Combinatorial Pattern Matching*, volume 3109 of *LNC3*, pages 400–408, 2004.
- [2] R. Baeza-Yates, B. Ribeiro-Neto, et al. *Modern information retrieval*, volume 463. ACM press New York, 1999.
- [3] B. B. Cambazoglu, V. Plachouras, and R. Baeza-Yates. Quantifying performance and quality gains in distributed web search engines. In *Proceedings of the 32nd international ACM SIGIR conference on Research and development in information retrieval*, pages 411–418. ACM, 2009.
- [4] C. Clarke, I. Soboroff, and N. Craswell. Gov2 test collection, 2004.
- [5] E. S. De Moura, C. F. dos Santos, D. R. Fernandes, A. S. Silva, P. Calado, and M. A. Nascimento. Improving web search efficiency via a locality based static pruning method. In *Proceedings of the 14th international conference on World Wide Web*, pages 235–244. ACM, 2005.
- [6] J. Dimond. Faster full text search through document clustering. diploma thesis, Karlsruhe Institute of Technology, 2013.
- [7] P. Elias. Universal codeword sets and representations of the integers. *Information Theory, IEEE Transactions on*, 21(2):194–203, 1975.
- [8] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner. SAP HANA Database: Data management for modern business applications. *SIGMOD Rec.*, 40(4):45–51, Jan. 2012.
- [9] S. W. Golomb. Run-length encodings. *IEEE Trans Info Theory*, 12(3):399–401, 1966.
- [10] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.
- [11] C. D. Manning, M. Surdeanu, J. Bauer, J. Finkel, S. J. Bethard, and D. McClosky. The Stanford CoreNLP natural language processing toolkit. In *Proceedings of 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pages 55–60, 2014.

- [12] G. Pass, A. Chowdhury, and C. Torgeson. A picture of search. In *Proceedings of the 1st international conference on Scalable information systems*, page 1. Citeseer, 2006.
- [13] M. Persin. Document filtering for fast ranking. In *Proceedings of the 17th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 339–348. Springer-Verlag New York, Inc., 1994.
- [14] P. Sanders and F. Transier. Fast intersection in randomized inverted indices. In *Workshop on Algorithm Engineering and Experiments (ALENEX)*, 2007.
- [15] F. Transier and P. Sanders. Engineering basic algorithms of an in-memory text search engine. *ACM Trans. Inf. Syst.*, 29(1), 2010.
- [16] C. Van Rijsbergen. *Information retrieval*. Butterworths, 1979.

A Data Compression

Figure 8 shows the space consumption per posting list entry using several encoding techniques. An interesting observation is that Golomb coding [9] is best without clustering whereas Elias- γ/δ coding [7, 2] is better with clustering. The reason is that these encoding schemes can better adapt to varying distances. Considering that data compression was not our primary objective, the savings are considerable.

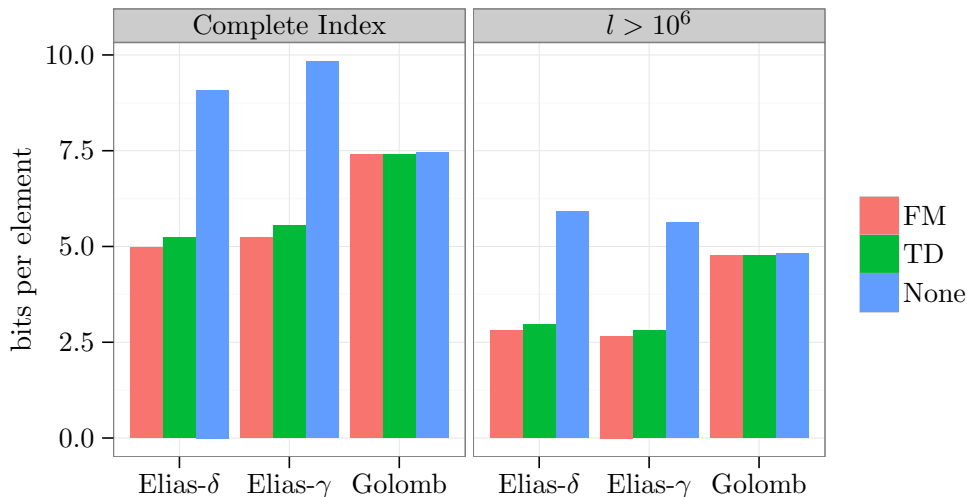


Figure 8: Compression of the inverted index using different clustering algorithms and encodings on the GOV2 dataset with $k = 1280$ clusters.

B Comparison Based Intersection

Figure 9 compares the theoretical speedup obtained using the cost function $\Phi(x, y) = \min(x, y)$ (S_T) and using the function stemming from comparison based list intersection [1] (S_L). We see

that for a comparison based list intersection algorithm, even higher speedups are predicted.

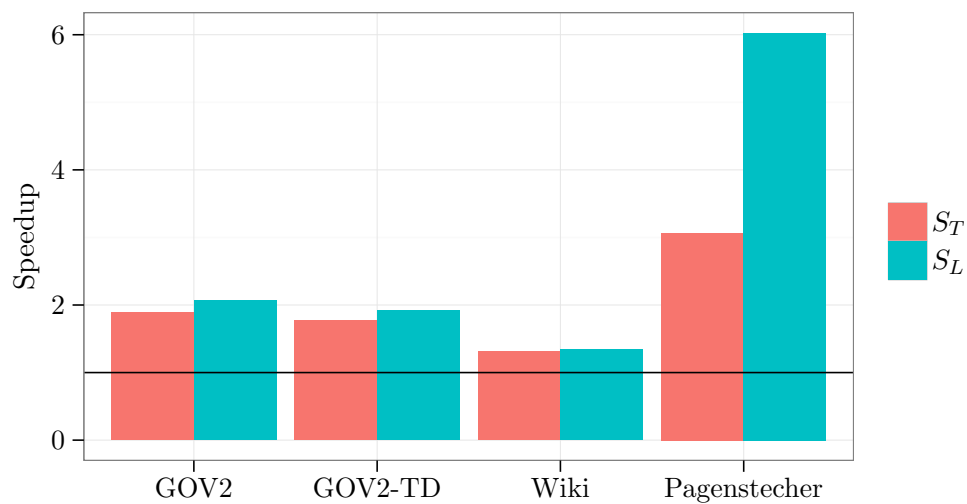


Figure 9: Speedups S_T and S_L on different datasets using the adapted cost function for S_L . The clusterings have $K = 1280$ clusters and use the FMClustering algorithm with exception of GOV2-TD.